

# WBIO Projekt

Kacper Średnicki

Politechnika Warszawska, Cyberbezpieczeństwo

20 kwietnia 2024

## Spis treści

<b>1. Wstęp</b>	2
<b>2. Generowanie liczb losowych z ruchu bakterii</b>	2
2.1. TRNG - charakterystyka i zastosowanie w projekcie	2
2.2. Rozważane i odrzucone koncepcje rozwiązania	2
2.2.1. Generator zaproponowany na etapie PitStop I	2
2.2.2. Camera noises generator	2
2.2.3. Audio & video generator	2
2.3. Zaimplementowane rozwiązanie	2
2.3.1. Chaotic map	2
2.3.2. Koncepcja generatora	3
2.3.3. Implementacja	3
2.3.4. Testy	6
<b>3. Podsumowanie</b>	9
<b>4. Bibliografia</b>	10

## **1. Wstęp**

Niniejszy dokument jest opisem postępów pracy nad projektem z przedmiotu Cyberbiobezpieczeństwo na dzień 20 kwietnia 2024 roku (PitStop II).

## **2. Generowanie liczb losowych z ruchu bakterii**

### **2.1. TRNG - charakterystyka i zastosowanie w projekcie**

Generowanie liczb losowych jest kluczowym elementem wybranego rozwiązania. To od jego jakości zależy bezpieczeństwo uzgodnienia i finalnej komunikacji. W celu osiągnięcia jak najwyższego poziomu losowości generatora liczb, zdecydowaliśmy się na zaimplementowanie tzw. TRNG (True Random Number Generator), nazywanego także generatorem niedeterministycznym. Sformułowanie to określa generator, którego działanie oparte jest na istniejącym zjawisku fizycznym. Takie zjawisko, dzięki swojej nieprzewidywalności, staje się źródłem entropii generatora TRNG. Do powyższego opisu doskonale zdaje się pasować proces generowania liczb losowych na podstawie ruchu bakterii, opisywany już na wcześniejszych etapach projektu. Rozważyliśmy kilka koncepcji stworzenia takiego generatora, a następnie podjęliśmy się implementacji naszym zdaniem najciekawszej opcji.

### **2.2. Rozważone i odrzucone koncepcje rozwiązania**

#### **2.2.1. Generator zaproponowany na etapie PitStop I**

Na etapie PitStopu nr 1 opisaliśmy wstępnie zaimplementowany przez nas algorytm, generujący liczby losowe na podstawie filmu symulującego obraz poruszającej się bakterii rejestrowany przez mikroskop. Klatki filmu zamieniane są na poziomy szarości, a następnie obliczane różnice poszczególnych pikseli. Zrezygnowaliśmy z wykorzystania tego algorytmu w projekcie, ze względu na jego zbyt trywialny charakter oraz ograniczone możliwości dostosowania do naszych potrzeb.

#### **2.2.2. Camera noises generator**

Drugim rozważonym przez nas algorytmem był algorytm, wykorzystujący szумy obrazu cyfrowego aparatu fotograficznego przy zmieniających się warunkach oświetleniowych [2]. Co prawda zmienne oświetlenie nie jest elementem powiązanym z naszą koncepcją projektową, ale mimo to zdecydowaliśmy się przetestować czułość generatora na zmiany jasności klatek, wywołane przez poruszające się bakterie. Zaimplementowaliśmy w tym celu wstępnią wersję algorytmu w języku Python i przetestowaliśmy jego działanie. Okazało się, że generowane były liczby ze stosunkowo niewielkiego przedziału, a także gołym okiem widać było często powtarzające się wartości. Algorytm wymagałby rozbudowania o zaimplementowane sztuczne generatory szumu, czy przesunięcia pikseli. Wątpliwy stały się wtedy jednak niedeterministyczny charakter generatora, zatem koncepcja ta została przed nas odrzucona.

#### **2.2.3. Audio & video generator**

Innym z pozoru interesującym algorytmem wydawał się algorytm generujący liczby losowe na podstawie audio i wideo [1]. Jako, że naszym założeniem projektowym jest generowanie liczb z ruchu bakterii, naturalnie bylibyśmy zmuszeni do implementacji wyłącznie funkcjonalności generowania liczb z wideo. Algorytm zakładał jednak analizowanie zaledwie 15-20 klatek filmu, ponieważ to generowanie na podstawie audio odgrywało w nim kluczową rolę. Sprawiło to, że ten algorytm również został przez nas odrzucony.

## **2.3. Zaimplementowane rozwiązanie**

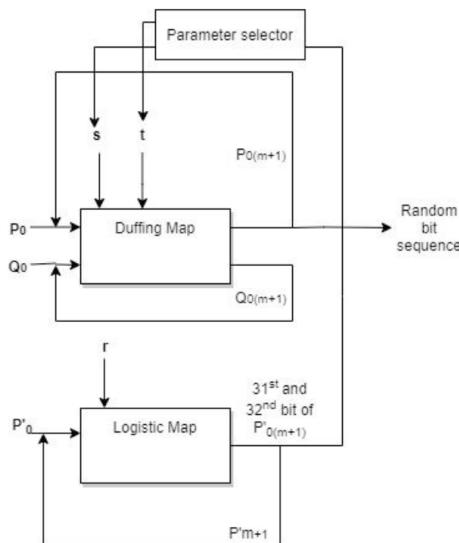
### **2.3.1. Chaotic map**

Algorytm umożliwiający generowanie liczb losowych, który wybraliśmy do zastosowania w projekcie, jest oparty na tzw. mapach chaotycznych. Mapa chaotyczna jest funkcją ewolucji, używaną do tworzenia dyskretnych układów dynamicznych. Wykazuje ona pewien chaotyczny charakter. Zgodnie z teorią chaosu, układy chaotyczne są ściśle zależne od warunków początkowych, a ich niewielka zmiana prowadzi do nieproporcjonalnie dużych zmian na wyjściu układu. Istnieje wiele typów map chaotycznych, w projekcie zastosowaliśmy dwa

z nich. Głównym zastosowaniem map chaotycznych w kryptografii jest szyfrowanie i deszyfrowanie obrazów. Można je jednak wykorzystać również do generowania liczb losowych. Algorytmy generujące oparte na mapach chaotycznych charakteryzują się wysokim poziomem losowości, zarówno dla modeli niedeterministycznych, jak i deterministycznych. Z tego powodu, zdecydowaliśmy się na wykorzystanie map chaotycznych do stworzenia generatora liczb losowych, opartego na ruchu bakterii.

### 2.3.2. Koncepcja generatora

Po przeanalizowaniu potencjalnych, pozytywnych aspektów zastosowania map chaotycznych w naszym generatorze, stworzyliśmy koncepcję naszego rozwiązania. Połączenie niedeterministycznego źródła entropii w postaci poruszającej się bakterii, z deterministycznym algorytmem opartym na chaosie, wymagało wybrania odpowiedniego algorytmu w celu zapewnienia tej drugiej części składowej. Zdecydowaliśmy się na implementację algorytmu generowania liczb losowych z wykorzystaniem mapy chaotycznej Duffinga oraz logistycznej mapy chaotycznej [8]. Schemat blokowy algorytmu, na którym w pewnym stopniu się wzorowaliśmy, został przedstawiony na rysunku 1. Natomiast bardziej szczegółowy opis tego algorytmu oraz sposobu połączenia go ze źródłem entropii został przedstawiony w sekcji 2.3.3.



Rys. 1: Schemat algorytmu opartego na mapie Duffinga i mapie logistycznej [8]

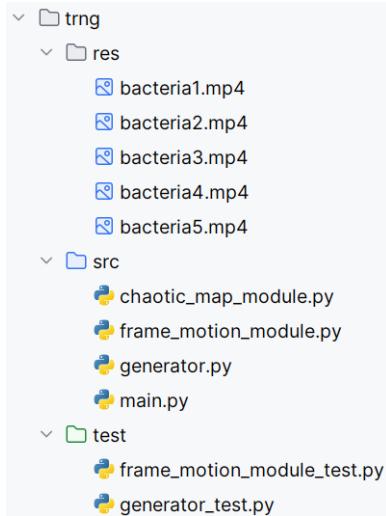
Generatory TRNG wykorzystujące mapy chaotyczne charakteryzują się także dużą wydajnością. Istniejące sprzętowe implementacje takich generatorów osiągają szybkość generacji nawet w granicach  $400Mbps$ . Biorąc pod uwagę całokształt koncepcji naszego projektu (wykorzystanie pojedynczej wygenerowanej liczby losowej jako klucz prywatny w protokole Diffiego-Hellmana), duża ilość liczb wygenerowanych w jednostce czasu nie jest kluczowym elementem. Najważniejszą cechą generatora w naszym rozwiążaniu jest natomiast faktyczny poziom losowości generowanych liczb. Z tego powodu zdecydowaliśmy, że nasz generator generował będzie jedną liczbę losową z każdą kolejną klatką filmu.

### 2.3.3. Implementacja

Na potrzeby korelacji pomiędzy naturalnym źródłem entropii i wybranym algorytmem, musieliśmy dokonać pewnych modyfikacji w przedstawionym wyżej algorytmie. Stałe parametry wykorzystywane w równaniach map chaotycznych, stanowiące swego rodzaju ziarno proponowanego generatora, zostały zastąpione przez dynamicznie generowane i obliczane wartości.

Zaimplementowany generator składa się z trzech głównych modułów. Moduły `frame_motion_module` oraz `chaotic_map_module` realizują na swój, odmienny sposób funkcjonalność generowania liczb losowych z ruchu bakterii. Oryginalny algorytm oparty na mapach chaotycznych zakładał losowe wybieranie współczynników, wykorzystywanych w równaniach map chaotycznych, ze stałego zdefiniowanych w listach tablicach. My zdecydowaliśmy się generować zawartość tych list również z ruchu bakterii. Naturalne źródło entropii jest zatem osiągane w naszym generatorze z wykorzystaniem dwóch metod. Moduł `generator` natomiast, odpowiedzialny jest za

połączenie w całość dwóch wspomnianych modułów i wykonywanie ich funkcjonalności cyklicznie, dla każdej klatki. Moduły reprezentowane są przez analogicznie nazwane skrypty w języku Python (rys. 2).



Rys. 2: Zaimplementowane moduły generatora

Poniżej zostały opisane szczegółowo implementacyjne trzech głównych modułów:

a) **Moduł frame\_motion\_module**

Jest to moduł odpowiedzialny za wygenerowanie z ruchu bakterii trzech list liczb losowych dla danej klatki, wykorzystywanych później w module `chaotic_map_module`. Główną funkcją modułu `frame_motion_module` jest funkcja `process_frame_motion()` przyjmująca jako parametry: aktualną klatkę filmu, poprzednią klatkę w skali szarości, listę wygenerowanych i niewykorzystanych liczb (wyjaśnione w dalszej części), wartość czułości detekcji ruchu, a także minimalny rozmiar obszaru ruchu, który ma być brany pod uwagę. Do wykonania poszczególnych operacji wykorzystuje ona różne funkcje z biblioteki `opencv` [7].

Na początku klatka konwertowana jest na obraz szary, dzięki czemu znana jest skala szarości danej klatki. Obliczana jest wartość bezwzględna różnicy pomiędzy szarościami aktualnej i poprzedniej klatki. Wartość ta jest następnie progowana zgodnie z podaną do funkcji wartością czułości. Na podstawie progowanej klatki, funkcja znajduje kontury (obszary odrębnego ruchu) używając funkcji `cv2.findContours()`. Obszary ruchu, których powierzchnia przekracza zdefiniowany przez użytkownika próg powierzchni wyrażony w pikselach, są dodawane do listy obszarów ruchu `motion_areas` i zaznaczane na wyświetlanej użytkownikowi oryginalnej klatce (wokół obszarów ruchu automatycznie rysowane są prostokąty, wizualizujące użytkownikowi miejsca i ilość wykrywanych przez program obszarów). Przykładowe klatki z zaznaczonymi przez program obszarami ruchu przedstawione zostały w sekcji 2.3.4. Po wykonaniu tych kroków funkcja `generate_list_of_numbers_from_motion_areas()` generuje listę liczb losowych na podstawie obszarów ruchu: iteruje po każdym obszarze ruchu i oblicza dla niego sumę wartości odcienni szarości pikseli (wykorzystując do tego funkcję `numpy.sum()` [6]), odpowiednio ją skalując. Fragment funkcji `process_frame_motion()`, implementujący opisane wyżej operacje, został przedstawiony na rysunku 3.

```

# konwersja klatki na obraz w skali szarości
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
# obliczenie różnicy między klatkami
diff = cv2.absdiff(prev_gray, gray)
# progowanie obrazu różnicy
_, thresh = cv2.threshold(diff, sensitivity, 255, cv2.THRESH_BINARY)
# znalezienie konturów na obrazie progowanym
contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
                               cv2.CHAIN_APPROX_SIMPLE)
motion_areas = []
# iteracja po konturach
for contour in contours:
    # jeśli obszar konturu jest wystarczająco duży, dodaj go do listy
    if cv2.contourArea(contour) > motion_area_size:
        motion_areas.append(contour)
# generowanie liczb losowych na podstawie obszarów ruchu
random_numbers = generate_list_of_numbers_from_motion_areas(motion_areas)

```

Rys. 3: Fragment funkcji `process_frame_motion()`

Aby nasza modyfikacja nie wykluczała się z dokumentacją [8], musielimy z każdej kolejnej klatki wygenerować trzy tablice składające się z czterech odpowiednio sformatowanych liczb. Jako, że ruch na filmie jest zmienny, ilości liczb wygenerowanych dla każdej kolejnej klatki różnią się od siebie. W celu zapewnienia modułowi `chaotic_map_module` trzech list z czerterema liczbami, musielimy znaleźć rozwiążanie sytuacji, w której zostało wygenerowanych mniej niż 12 liczb. Zaimplementowaliśmy w związku z tym mechanizm, zapisujący nadmiarowo wygenerowane wartości do listy `unused_numbers`. Przechowuje ona liczby, aby mogły zostać wykorzystane w przypadku niewystarczającej ilości wygenerowanych liczb dla danej klatki. Mechanizm kontroluje też maksymalny rozmiar listy, aby pamięć programu nie była zbytnio obciążana. Funkcja `process_frame_motion()` zwraca trzy sformatowane listy wygenerowanych liczb, aktualną klatkę w skali szarości oraz aktualną listę `unused_numbers`. Pełen kod implementujący moduł `frame_motion_module` jest dostępny tutaj (hasło: "wbio").

#### b) Moduł `chaotic_map_module`

Jest to moduł odpowiedzialny za wygenerowanie finalnej liczby losowej dla danej klatki. Jego główną funkcją jest funkcja `generate_number_chaotic_map()`. Przyjmuje ona następujące parametry: aktualną klatkę filmu, trzy tablice liczb (`r_values`, `s_values` oraz `t_values`, za których wygenerowanie odpowiedzialny jest moduł `frame_motion_module`), wartość kontrolną wykorzystywaną do odpowiedniego formatowania liczby, a także oczekiwana przez użytkownika liczbę bitów, na której ma być możliwe zapisanie wygenerowanej liczby.

Na początku wywoływana jest w niej funkcja `process_frame_select_parameters()`. Korzystając z operacji `numpy.mean()`, oferowanej przez bibliotekę *NumPy* [6], wylicza średnią wartość pikseli we wszystkich kanałach RGB (red, green, blue). W ten sposób generowana jest wartość liczbową dla danej klatki. Funkcja `process_frame_select_parameters()` po odpowiednim przeskalowaniu tej wartości, pobiera z niej dwa ostatnie bity. W kolejnym kroku, na podstawie utworzonej przez nie dwu-bitowej liczby binarnej, wyznacza indeks. Zgodnie z tym indeksem, wybiera następnie po jednej wartości z każdej z trzech list (rys. 4), wygenerowanych w module `frame_motion_module`.

```

# k - drugi najmłodszy bit, m - najmłodszy bit
k = get_second_last_bit(formatted_processed_data)
m = get_last_bit(formatted_processed_data)
if k == 0 and m == 0:
    r, s, t = r_values[0], s_values[0], t_values[0]
elif k == 0 and m == 1:
    r, s, t = r_values[1], s_values[1], t_values[1]
elif k == 1 and m == 0:
    r, s, t = r_values[2], s_values[2], t_values[2]
elif k == 1 and m == 1:
    r, s, t = r_values[3], s_values[3], t_values[3]

```

Rys. 4: Fragment funkcji `process_frame_select_parameters()`

Zwrócone wartości `r`, `s` oraz `t` są podstawiane przez funkcję `generate_number_chaotic_map()` do równań mapy Duffinga i logistycznej. Obliczone z tych równań wartości `Q` oraz `g` są odpowiednio formatowane (zgodnie z parametrem kontrolnym i podaną przez użytkownika liczbą bitów). Wreszcie, wykonywana jest operacja XOR pomiędzy sformatowanymi wartościami `Q` oraz `g`. Wynik tej operacji jest wygenerowaną liczbą losową. Poza tą liczbą, funkcja `generate_number_chaotic_map()` zwraca aktualne wartości `Q` oraz `g` tak, aby mogły one zostać w sposób rekurencyjny wykorzystane w równaniach map chaotycznych, przy kolejnym wywołaniu funkcji. Pełen kod implementujący moduł `chaotic_map_module` jest dostępny tutaj (hasło: "wbio").

### c) Moduł generator

Jest to moduł agregujący funkcjonalności modułów `frame_motion_module` oraz `chaotic_map_module`. W swojej funkcji `generate_random_numbers()` implementuje główną pętlę programu. Funkcja ta przyjmuje jako parametry: ścieżkę do pliku filmu, wartość czułości detekcji ruchu, minimalny rozmiar obszaru ruchu, który ma być brany pod uwagę oraz liczbę bitów, na której mogą być zapisane wygenerowane liczby losowe. Po inicjalizacji zmiennych następuje wykonanie pętli (rys. 5). W każdej jej iteracji, dla każdej kolejnej klatki filmu, zostają wykonane wszystkie opisane powyżej operacje. Funkcja zwraca listę wygenerowanych liczb losowych.

```
# wczytanie pliku video
cap = cv2.VideoCapture(video_path)

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    # wygenerowanie tablic r_values, s_values i t_values na podstawie ruchu
    lst1, lst2, lst3, prev_gray, unused_numbers = process_frame_motion(frame,
    ← prev_gray, unused_numbers, sensitivity, motion_area_size)
    # wygenerowanie finalnej liczby losowej
    random_number, Q, g = generate_number_chaotic_map(frame, lst1, lst2, lst3,
    ← Q, g, control_param, n_bits)
    # dodanie wygenerowanej liczby losowej do końcowej listy
    generated_numbers.append(random_number)
    # przerwanie głównej pętli programu przyciskiem 'q'
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
```

Rys. 5: Fragment funkcji `generate_random_numbers()`

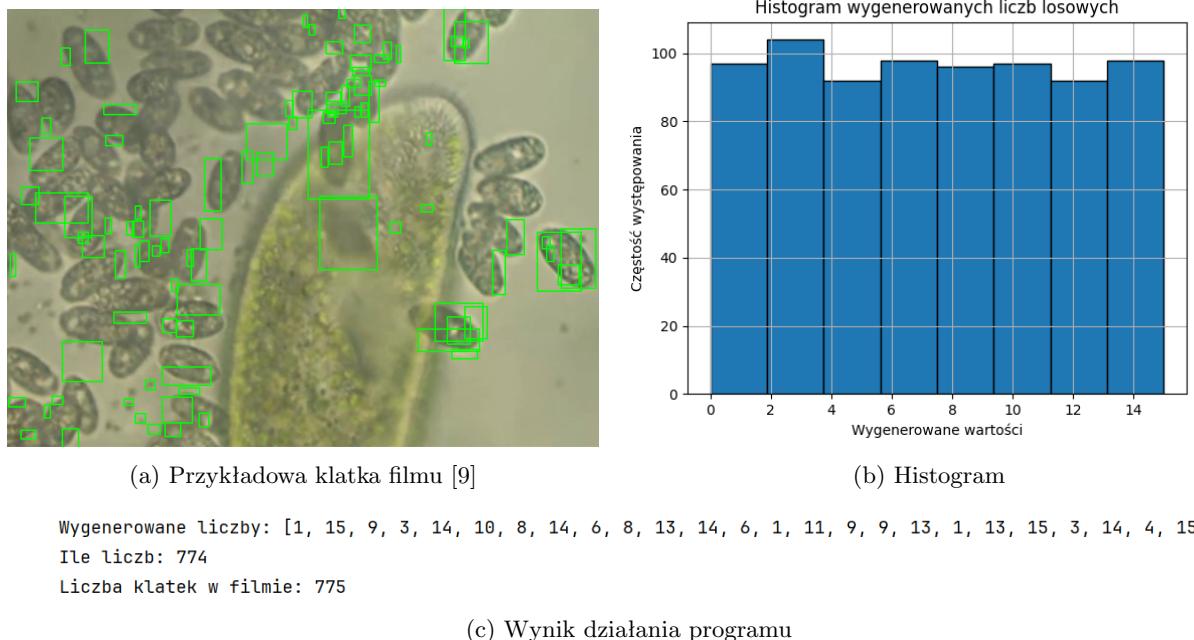
Na potrzeby naszego projektu przygotowana została też bardzo podobna funkcja `generate_single_number()`, generująca pojedynczą liczbę losową. Może ona zostać przez nas wykorzystana jako klucz prywatny w protokole Diffiego-Hellmana. Pełen kod implementujący moduł `generator` jest dostępny tutaj (hasło: "wbio").

#### 2.3.4. Testy

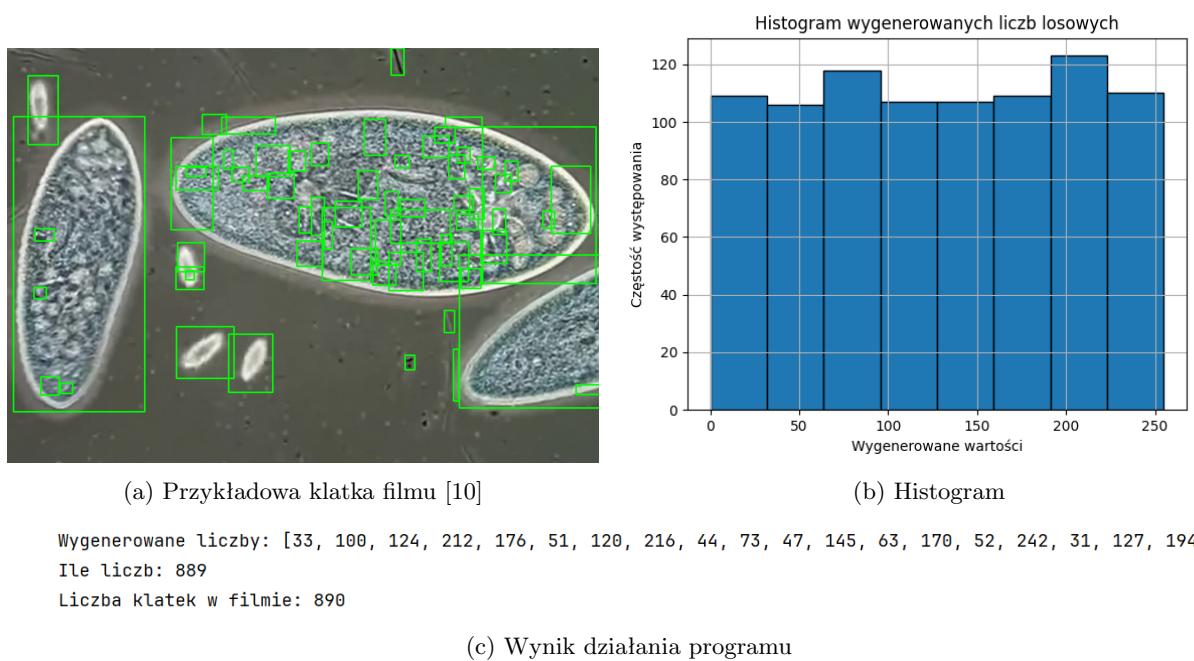
Biorąc pod uwagę to, że wysoki faktyczny poziom losowości generowanych liczb jest absolutnym priorytetem w naszym projekcie, zdecydowaliśmy się poświęcić dużo uwagi tematowi testowania losowości generowanych liczb.

Kluczowym aspektem w temacie testowania generowanych liczb jest fakt, że nie istnieje metoda jednoznacznie stwierdzająca pełną losowość generatora. Nie oznacza to jednak, że nie należy testować generatorów. Istnieje wiele różnych technik i narzędzi sprawdzających statystyczną losowość ciągów. Pozwalają one wykryć potencjalne wady generatora i tym samym wykazać jego niską jakość. Należy jednak pamiętać, że prawdziwe bezpieczeństwo kryptograficzne dotyczy tego, czy wynik generatora liczb losowych mógłby być przewidywalny dla bystrego atakującego, znającego szczegóły dotyczące zastosowanego algorytmu. Należy więc zawsze analizować losowość generatora pod kątem jakości źródła entropii. W naszym przypadku, nieprzewidywalny ruch bakterii wydaje się być jakościowym źródłem entropii [4].

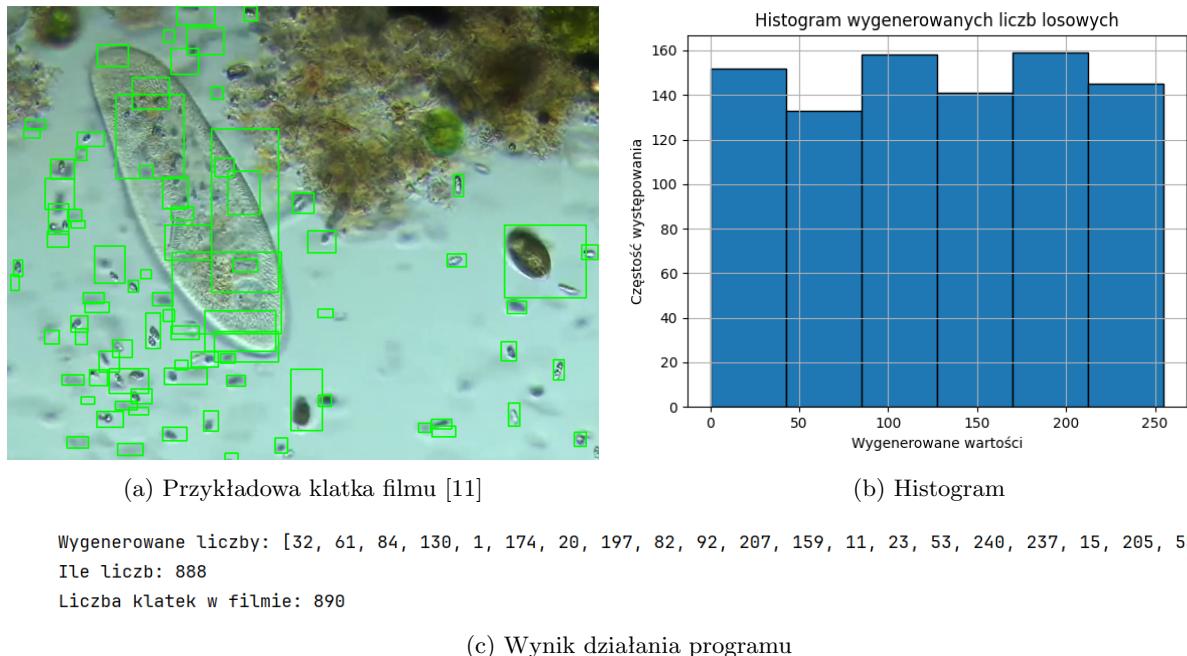
Na tym etapie, obserwowaliśmy działanie i jakość generatora własnymi, trywialnymi metodami. Przygotowaliśmy skrypt testowy `generator_test.py`, który poza wywołaniem generacji liczb losowych, sporządza histogram wygenerowanych wartości. Pełen kod tego skryptu jest dostępny tutaj (hasło: "wbio"). Dzięki temu, mogliśmy "na oko" przetestować nasz generator, poszukując potencjalnie niepokojących oznak w wyglądzie histogramów. Na rysunkach 6, 7, 8, 9 i 10 przedstawione zostały zrzuty z ekranu, dokumentujące wyniki części z przeprowadzonych przez nas testów. Wykorzystaliśmy nagrania obserwowanych pod mikroskopem bakterii, pierwotniaków i bezkręgowców.



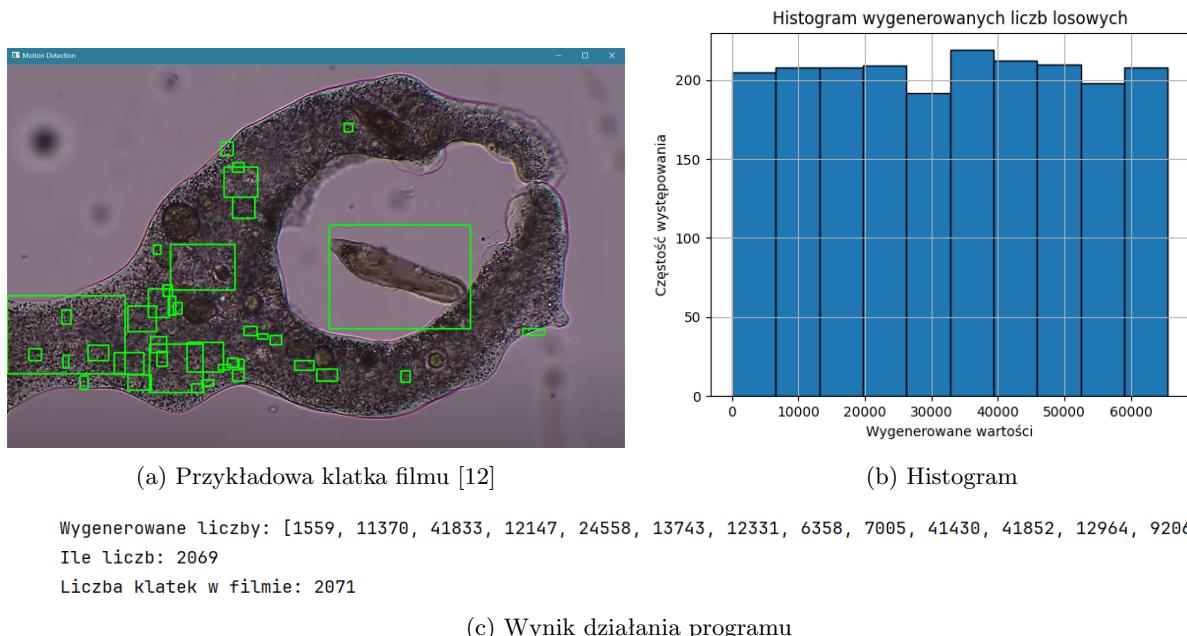
Rys. 6: Generacja liczb 4-bitowych (czułość: 12, min rozmiar wykrywanego obszaru ruchu: 80)



Rys. 7: Generacja liczb 8-bitowych (czułość: 10, min rozmiar wykrywanego obszaru ruchu: 150)



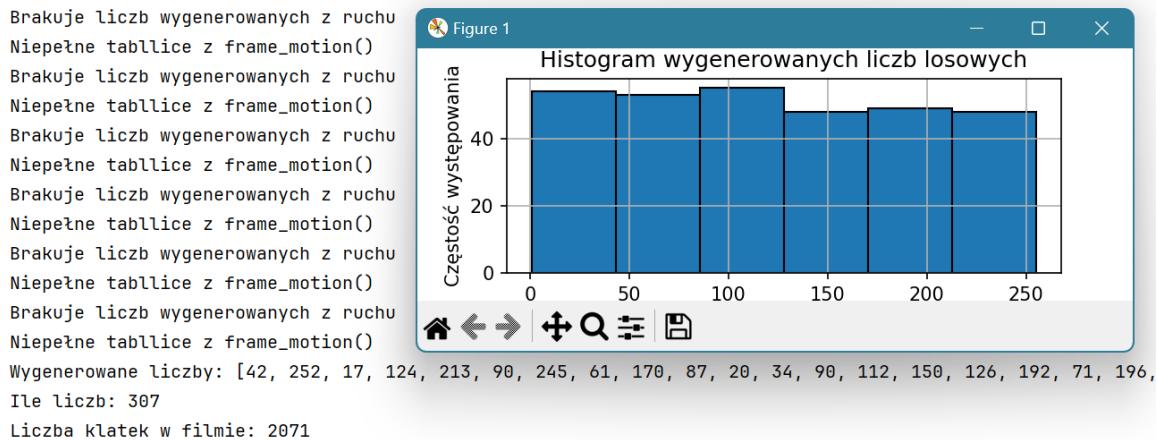
Rys. 8: Generacja liczb 8-bitowych (czułość: 15, min rozmiar wykrywanego obszaru ruchu: 200)



Rys. 9: Generacja liczb 16-bitowych (czułość: 5, min rozmiar wykrywanego obszaru ruchu: 50)

Jak widać na przedstawionych powyżej wynikach testów, liczby generowane są poprawnie na podstawie różnych filmów. Histogramy dowodzą, że liczby generowane są równomiernie, w ramach wskazanej przez użytkownika liczby bitów. W ogólności, przy optymalnie dobranych parametrach, generowanych jest  $n - 1$  liczb (gdzie  $n$  to liczba klatek). Przypadek, w którym wygenerowano o kilka liczb mniej niż  $n$ , oznacza najczęściej niewielki ruch na pierwszych klatkach obrazu, co skutkuje brakiem wypełnienia liczbami losowymi list w module `frame_motion_module` (opisany w 2.3.3).

Na rysunku 10 przedstawiony został wynik testu wykonanego dla nieoptymalnie dobranych parametrów.



Rys. 10: Generacja liczb przy nieodpowiednio dobranych parametrach (czułość: 10, min rozmiar obszaru: 100)

Jak widać na powyższym zrzucie z ekranu, przy nieoptymalnie dobranych parametrach wygenerowanych zostało znacznie mniej liczb, niż liczba klatek filmu. Mimo to, uzyskane wartości wygenerowane zostały równomiernie na zdefiniowanym przez użytkownika przedziale. Ten przypadek testowy udowadnia, że generator działa w sposób ciągły i oczekuje na kolejne przedziały czasowe, w których ruch zostanie wykryty. Jest to cecha zgodna z początkowymi założeniami - w przypadku obserwowania prawdziwego organizmu pod mikroskopem, w sposób naturalny mogą nastąpić momenty o niewielkim ruchu, które nie powinny powodować zakończenia pracy generatora.

Inną cechą naszego generatora, udowodnioną podczas testów, jest duża wrażliwość na warunki początkowe, charakteryzująca systemy oparte na układach chaotycznych (w naszym przypadku są nimi mapy chaotyczne). Jako, że źródłem entropii stworzonego generatora jest przedstawiony na filmie ruch organizmu żywego, zmiana filmu wybranego do generacji skutkuje otrzymaniem wyraźnie innej sekwencji wartości, o odmiennym rozkładzie. Podobnie, nawet niewielka modyfikacja parametrów wejściowych generatora prowadzi do otrzymania innych wyników.

Przeprowadzone testy i obserwacje nie wykazały żadnych niepokojących sygnałów w kontekście bezpieczeństwa generowanych liczb. Wszystko wskazuje na to, że zaimplementowany generator charakteryzuje się wysokim poziomem losowości. Sprawia to, że na tym etapie jesteśmy zadowoleni z jego działania. Mimo to, do czasu zakończenia projektu chcielibyśmy wyszukać i przeprowadzić także profesjonalne testy statystyczne wygenerowanych ciągów liczb.

Szczególnie interesującym rozwiązaniem wydaje się być przygotowane przez NIST oprogramowanie testowe [5], oferujące kilkanaście profesjonalnych typów testów losowości generowanych ciągów. NIST udostępnia normy, które powinny być spełnione dla poszczególnych testów. Poddanie naszego generatora takim testom z pewnością pozwoliłoby zidentyfikować, lub wykluczyć potencjalne błędy związane z losowością. Problematicznym elementem wydaje się być jednak duża ilość wygenerowanych danych, wymagana do przeprowadzenia testów z wykorzystaniem oprogramowania NIST [3]. W przypadku naszego rozwiązania, generującego jedną liczbę decymalną co klatkę filmu, dostarczenie do oprogramowania wymaganej ilości danych może okazać się bardzo czasochłonne, a także wymagające zapewnienia bardzo długiego filmu wykorzystywanego do generacji. Podejmiemy jednak próbę wykonania testów oferowanych przez NIST, a w przypadku niepowodzenia przetestujemy nasz generator samodzielnie, na większą skalę.

### 3. Podsumowanie

Podsumowując, projekt przeszedł przez istotne etapy rozwoju i analizy pomysłów, które wskazywały na różne koncepcje generowania liczb pseudolosowych z różnych źródeł biologicznych. Zbliżając się do ostatniego etapu, planujemy również rozwinać protokół wymiany kluczy Diffiego-Hellmana, aby w pełni wykorzystać potencjał generowanych liczb losowych opartych na ruchu bakterii. W procesie wymiany informacji poprzez nasz zmodernizowany protokół Diffiego-Hellmana, strony komunikujące się będą korzystać z kluczy kryptograficznych

opartych na ruchu bakterii do ustalenia wspólnego sekretnego klucza. Ten klucz będzie stanowił podstawę do bezpiecznego szyfrowania i deszyfrowania przesyłanych danych, zapewniając wysoki poziom poufności i integralności komunikacji.

#### 4. Bibliografia

- [1] I-Te Chen. *Random Numbers Generated from Audio and Video Sources*. Hindawi Publishing Corporation. 2013. URL: <https://www.hindawi.com/journals/mpe/2013/285373/> (visited on 04/17/2024).
- [2] Rongzhong Li. *A True Random Number Generator Algorithm From Digital Camera Image Noise For Varying Lighting Conditions*. Departments of Computer Science and Physics, Wake Forest University. 2016. URL: [https://wakespace.lib.wfu.edu/bitstream/handle/10339/62642/Li\\_wfu\\_0248M\\_10943.pdf](https://wakespace.lib.wfu.edu/bitstream/handle/10339/62642/Li_wfu_0248M_10943.pdf) (visited on 04/17/2024).
- [3] NIST. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. 2010. URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf> (visited on 04/23/2024).
- [4] NIST. *Recommendation for the Entropy Sources Used for Random Bit Generation*. 2018. URL: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90B.pdf> (visited on 04/23/2024).
- [5] *NIST SP 800-22 Statistical Test Suite Software*. <https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software>. (Visited on 04/23/2024).
- [6] *NumPy documentation*. <https://numpy.org/doc/stable/reference/>. (Visited on 04/19/2024).
- [7] *Open Source Computer Vision*. <https://docs.opencv.org/4.x/index.html>. (Visited on 04/06/2024).
- [8] K Sathya et al. *Security Analyses of Random Number Generation with Image Encryption Using Improved Chaotic Map*. Procedia Computer Science. 2022. URL: <https://www.sciencedirect.com/science/article/pii/S1877050922021160> (visited on 04/17/2024).
- [9] *Wideo nr 1 - Delta Optical*. [https://www.youtube.com/watch?v=6o1M\\_d7zIeU](https://www.youtube.com/watch?v=6o1M_d7zIeU). (Visited on 04/06/2024).
- [10] *Wideo nr 2 - EVIDENT Life Science*. <https://youtu.be/MxbwiACd0Tw?si=WCkc4N6pYk1Bt6Ki>. (Visited on 04/26/2024).
- [11] *Wideo nr 3 - Jan van Gastel's videos*. [https://youtu.be/qNNyjbwfeSo?si=\\_27r4ncPY15inZYk](https://youtu.be/qNNyjbwfeSo?si=_27r4ncPY15inZYk). (Visited on 04/26/2024).
- [12] *Wideo nr 4 - mantismundi*. <https://youtu.be/4XlzCe5gDu0?si=uMDGUW7ylfMqB66H>. (Visited on 04/26/2024).